

## SESSION 4

---

# iOS DEVELOPMENT

BEAUTIFUL VIEWS WITH

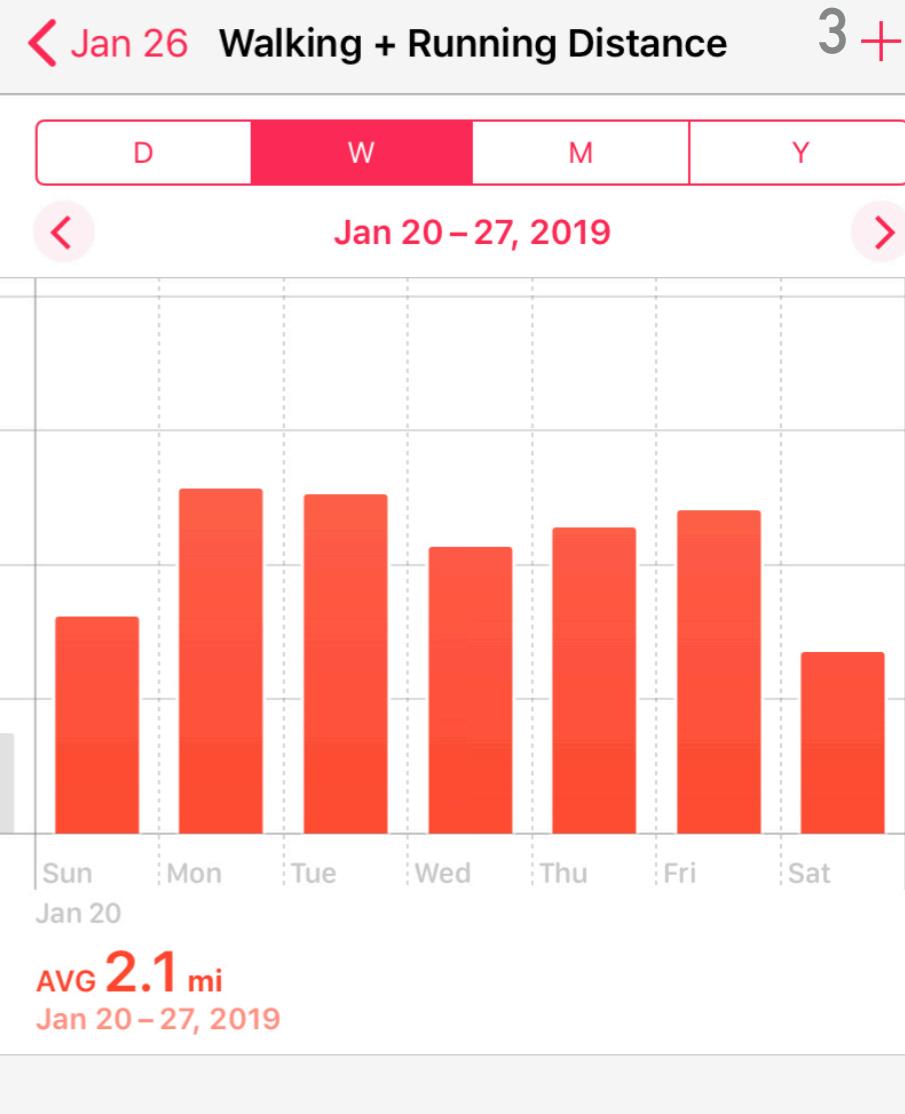
---

CUSTOM DRAWING

# WHY CUSTOM DRAWING?

- ▶ What are some ways you could create the bar graph?
- ▶ Consider:
  - ▶ Orange bars
  - ▶ Grid background

Health

Add to Favorites Show All Data Data Sources & Access Unit  mi 

Today



Health Data



Sources



Medical ID

# WHY CUSTOM DRAWING?

- ▶ What's more complex about this one?

&lt; Jan 26

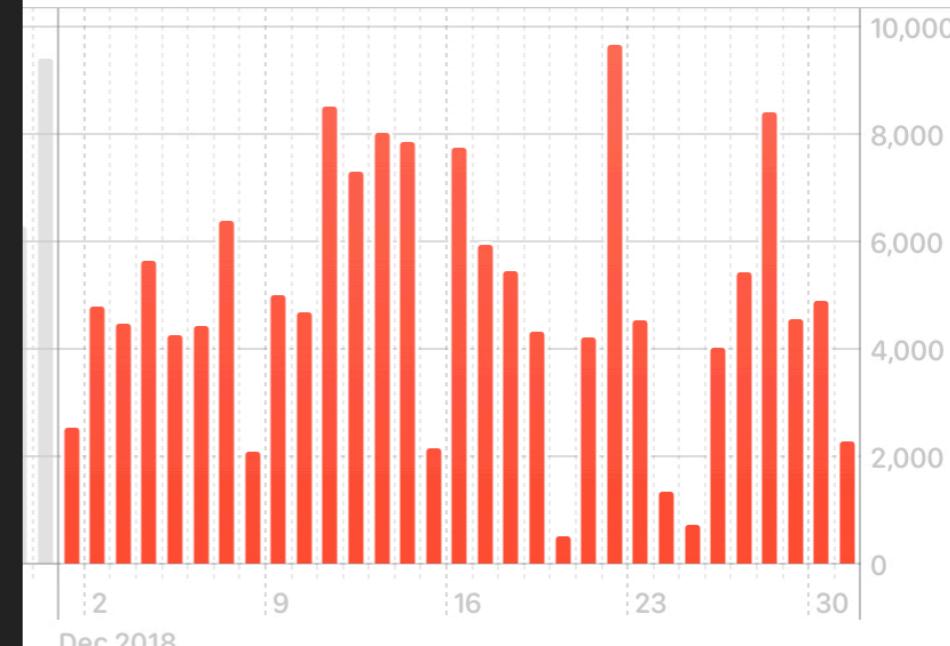
Steps

4 +

D W M Y



Dec 2018



Avg 4,895 steps  
Dec 2018

Add to Favorites 

Show All Data &gt;

Data Sources &amp; Access &gt;

Unit

Steps

Health



Today



Health Data



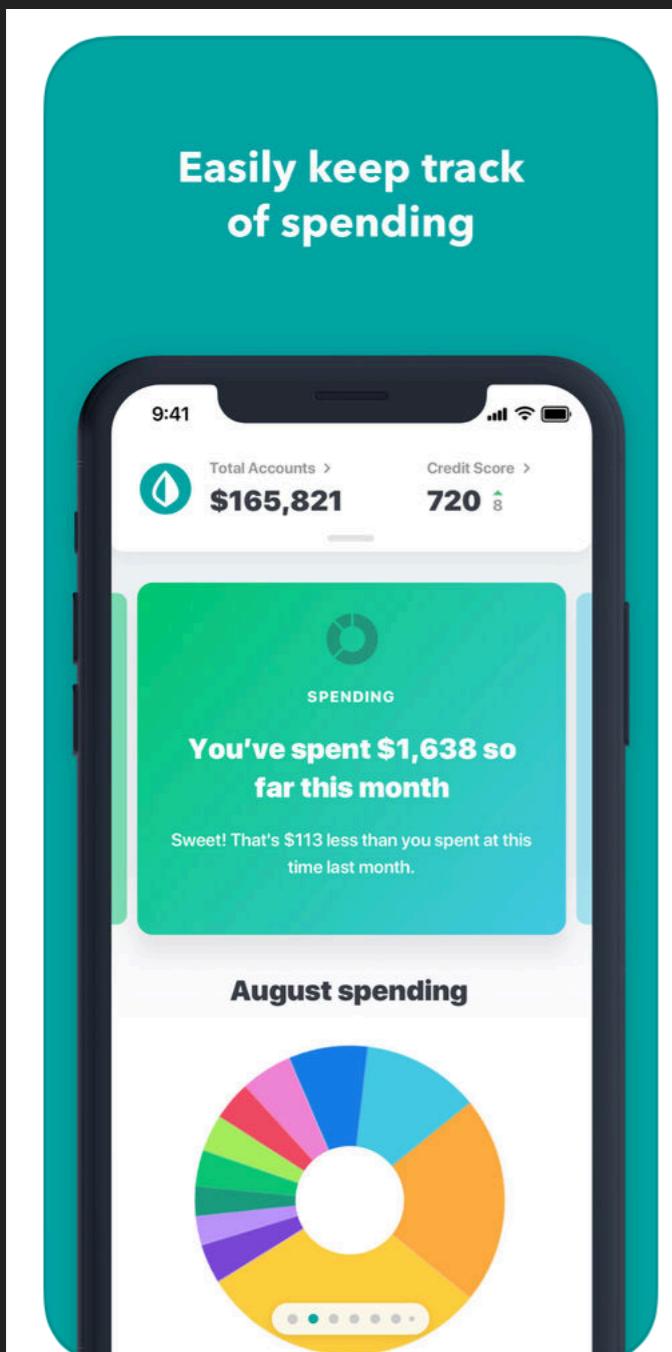
Sources



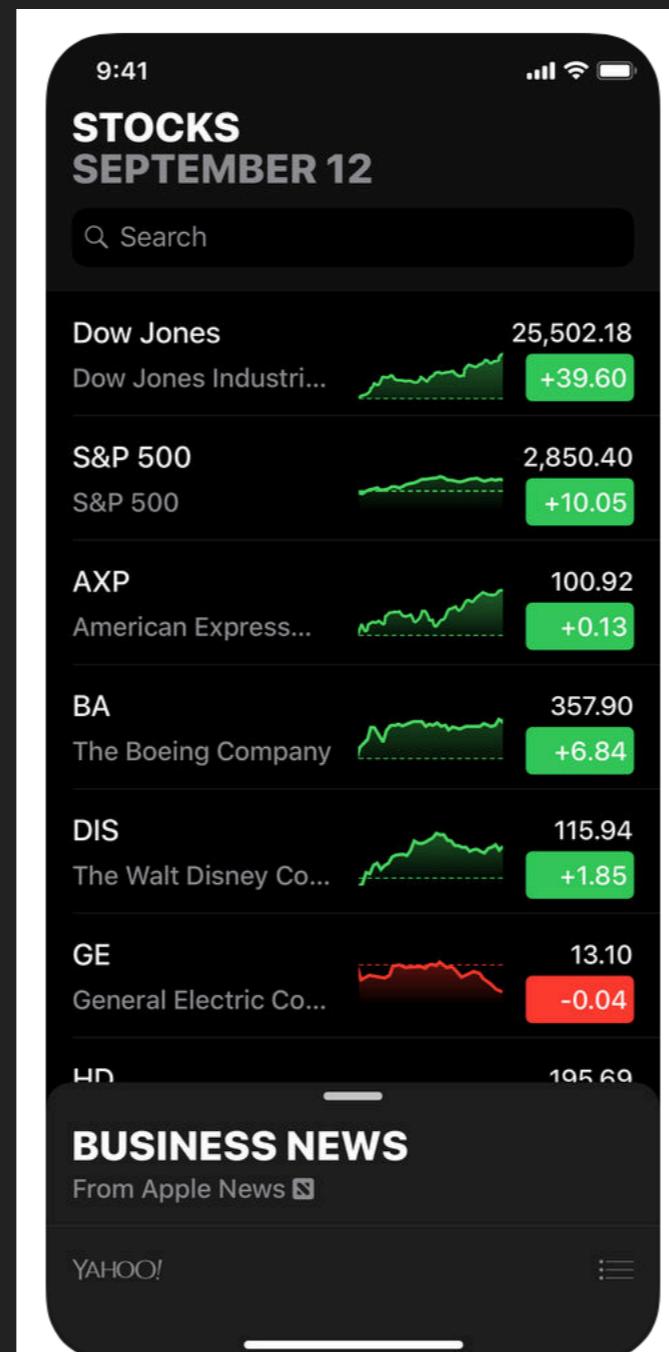
Medical ID

# EXAMPLES OF CUSTOM DRAWING

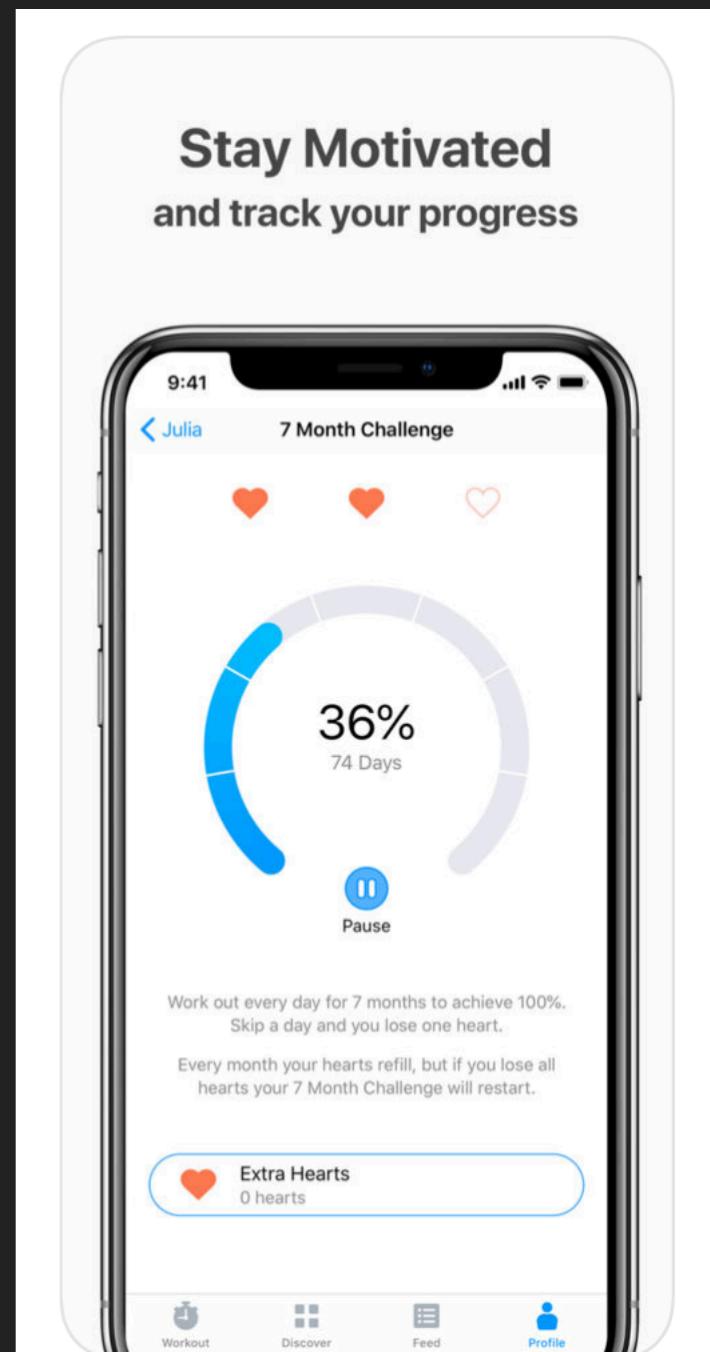
Mint



Stocks

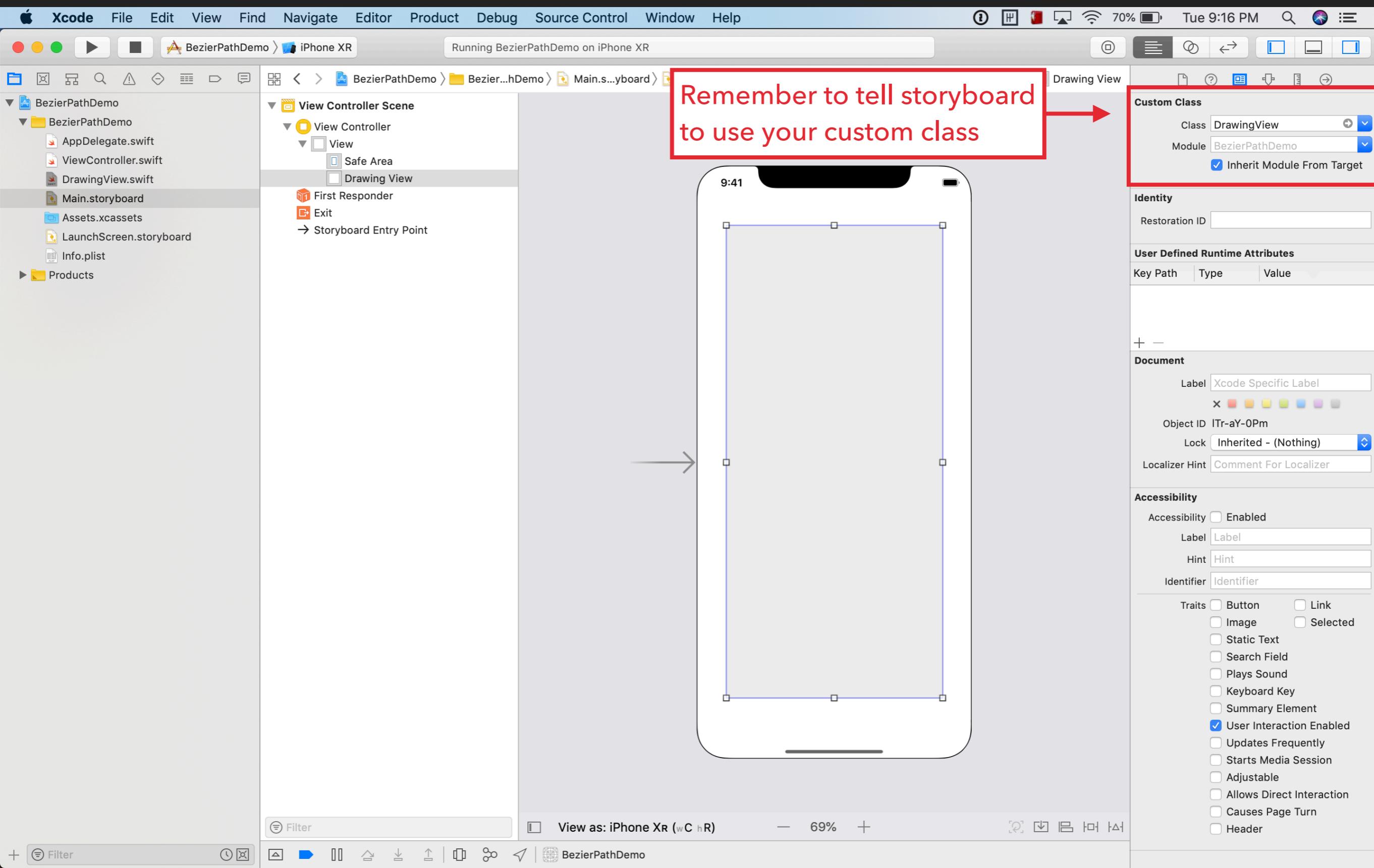


Seven



## GETTING STARTED

- ▶ Custom drawing always takes place inside a **UIView**
- ▶ You'll need to:
  - ▶ Create a subclass of **UIView**
  - ▶ Override the **draw** method



Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help

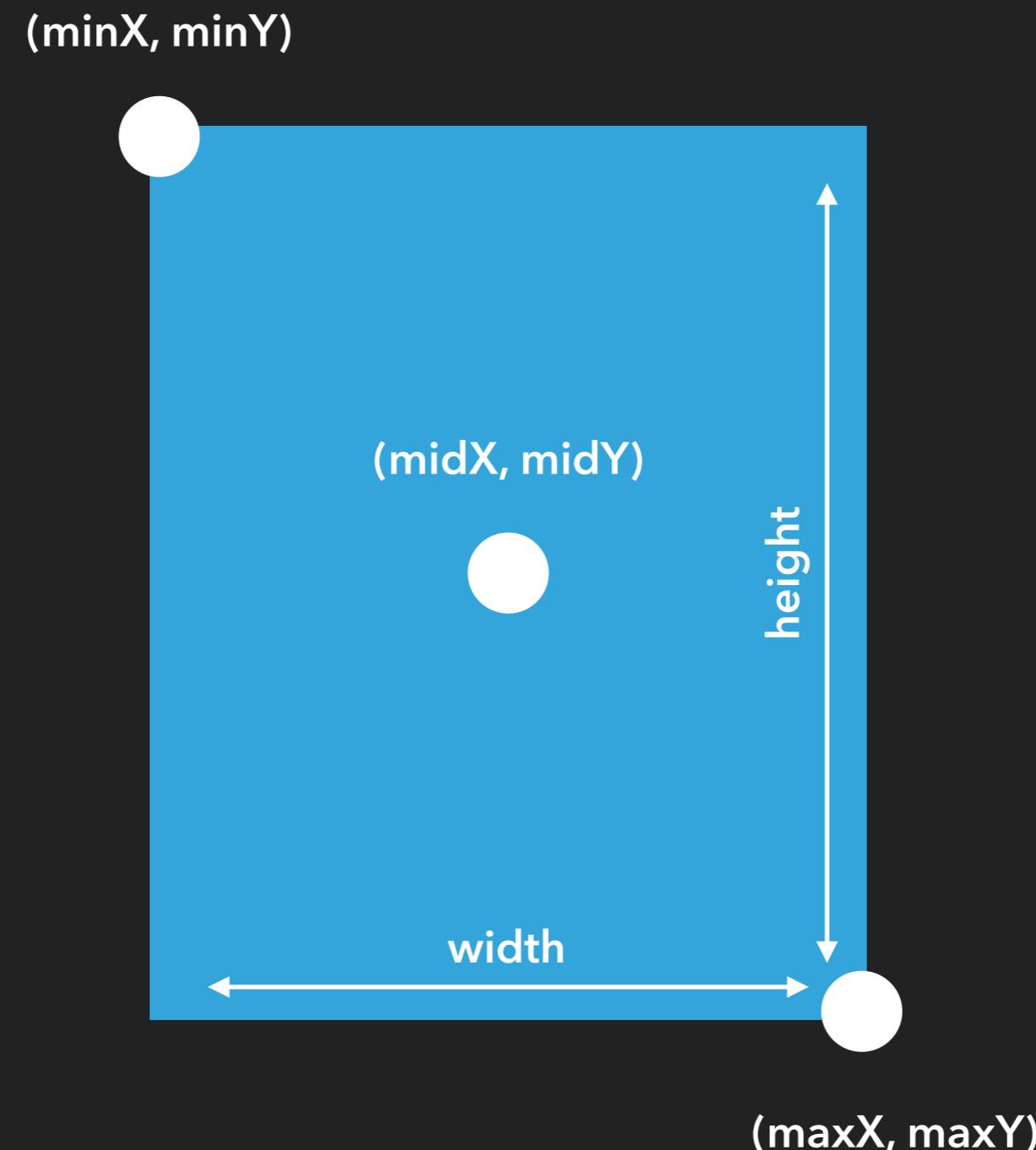
Session4Demos > iPhone 11 Pro Max Finished running Session4Demos on iPhone 11 Pro Max 5

Session4Demos Session4Demos Drawing DrawingView.swift No Selection

```
1 //  
2 // DrawingView.swift  
3 // Session4Demos  
4 //  
5 // Created by Susan Stevens on 1/25/20.  
6 // Copyright © 2020 Susan Stevens. All rights reserved.  
7 //  
8  
9 import UIKit  
10  
11 class DrawingView: UIView {  
12     override func draw(_ rect: CGRect) {  
13         // custom drawing code  
14     }  
15 }  
16 }  
17 }
```

rect defines the dimensions of your canvas

# PROPERTIES OF CGRECT



The screenshot shows the Xcode interface with the code editor and a running application in the simulator.

**Code Editor:**

```
4 //  
5 // Created by Susan Stevens on 1/25/20.  
6 // Copyright © 2020 Susan Stevens. All rights reserved.  
7 //  
8  
9 import UIKit  
10  
11 class DrawingView: UIView {  
12  
13     override func draw(_ rect: CGRect) {  
14         let width = rect.width  
15         let height = rect.height  
16  
17         // Draw dashed horizontal lines  
18         let lines = UIBezierPath() ← Create bezier path  
19  
20         lines.move(to: CGPoint(x: 0, y: height / 4))  
21         lines.addLine(to: CGPoint(x: width, y: height / 4))  
22  
23         lines.move(to: CGPoint(x: 0, y: 3 * height / 4))  
24         lines.addLine(to: CGPoint(x: width, y: 3 * height / 4))  
25  
26         lines.setLineDash([10, 5], count: 2, phase: 0)  
27         lines.lineWidth = 5  
28         lines.stroke()  
29  
30  
31  
32
```

**Simulator:**

The iPhone 11 Pro Max simulator displays two horizontal dashed lines at approximately one-quarter and three-quarters of the screen height. The status bar shows the time as 3:17. The navigation bar indicates "Running Session4Demos on iPhone 11 Pro Max". The bottom of the screen shows the Xcode toolbar with icons for Drawing, Layers, Animations 1, Gestures, and Animations 2.

The screenshot shows the Xcode interface with the code editor and a running application in the simulator.

**Xcode Editor:**

```
4 //  
5 // Created by Susan Stevens on 1/25/20.  
6 // Copyright © 2020 Susan Stevens. All rights reserved.  
7 //  
8  
9 import UIKit  
10  
11 class DrawingView: UIView {  
12  
13     override func draw(_ rect: CGRect) {  
14         let width = rect.width  
15         let height = rect.height  
16  
17         // Draw dashed horizontal lines  
18         let lines = UIBezierPath()  
19  
20         lines.move(to: CGPoint(x: 0, y: height / 4))  
21         lines.addLine(to: CGPoint(x: width, y: height / 4))  
22  
23         lines.move(to: CGPoint(x: 0, y: 3 * height / 4))  
24         lines.addLine(to: CGPoint(x: width, y: 3 * height / 4))  
25  
26         lines.setLineDash([10, 5], count: 2, phase: 0)  
27         lines.lineWidth = 5  
28         lines.stroke()  
29  
30  
31  
32
```

A red callout box highlights the code from line 20 to 21, with a red arrow pointing down to the iPhone simulator screen. A red box also highlights the text "Draw the top line".

**iPhone Simulator:**

The iPhone 11 Pro Max simulator displays a white screen with two horizontal dashed lines. The top dashed line is positioned at approximately y = height / 4 and the bottom one at approximately y = 3 \* height / 4. The simulator status bar shows the time as 3:17 and battery level as 23%.

**Bottom Bar:**

Drawing, Layers, Animations 1, Gestures, Animations 2

iPhone 11 Pro Max — 13.3

## MOVE VS. ADD LINE

- ▶ **move** means “put the pencil down at this point”
- ▶ **addLine** means “draw a line to this point ”

```
path.move(to: CGPoint(x: width / 2, y: 0))
path.addLine(to: CGPoint(x: width / 2, y: height))
```

# UIBezierPath

The screenshot shows the Xcode interface with the code editor and a running application in the simulator.

**Code Editor:**

```
4 //  
5 // Created by Susan Stevens on 1/25/20.  
6 // Copyright © 2020 Susan Stevens. All rights reserved.  
7 //  
8  
9 import UIKit  
10  
11 class DrawingView: UIView {  
12  
13     override func draw(_ rect: CGRect) {  
14         let width = rect.width  
15         let height = rect.height  
16  
17         // Draw dashed horizontal lines  
18         let lines = UIBezierPath()  
19  
20         lines.move(to: CGPoint(x: 0, y: height / 4))  
21         lines.addLine(to: CGPoint(x: width, y: height / 4))  
22  
23         lines.move(to: CGPoint(x: 0, y: 3 * height / 4))  
24         lines.addLine(to: CGPoint(x: width, y: 3 * height / 4))  
25  
26         lines.setLineDash([10, 5], count: 2, phase: 0)  
27         lines.lineWidth = 5  
28         lines.stroke()  
29  
30  
31  
32
```

A red box highlights the code from line 23 to 24, and a red callout box with the text "Draw the bottom line" points to the second line of this highlighted block. A red arrow points from the text to the code.

**Simulator:**

The iPhone 11 Pro Max simulator displays two horizontal dashed lines at approximately one-quarter and three-quarters of the screen height. The status bar shows the time as 3:17. The navigation bar indicates the app is "Running Session4Demos on iPhone 11 Pro Max". The tab bar at the bottom includes "Drawing" (selected), "Layers", "Animations 1", "Gestures", and "Animations 2".

The screenshot shows the Xcode interface with the code editor and a running application in the simulator.

**Code Editor:**

```
4 //  
5 // Created by Susan Stevens on 1/25/20.  
6 // Copyright © 2020 Susan Stevens. All rights reserved.  
7 //  
8  
9 import UIKit  
10  
11 class DrawingView: UIView {  
12  
13     override func draw(_ rect: CGRect) {  
14         let width = rect.width  
15         let height = rect.height  
16  
17         // Draw dashed horizontal lines  
18         let lines = UIBezierPath()  
19  
20         lines.move(to: CGPoint(x: 0, y: height / 4))  
21         lines.addLine(to: CGPoint(x: width, y: height / 4))  
22  
23         lines.move(to: CGPoint(x: 0, y: 3 * height / 4))  
24         lines.addLine(to: CGPoint(x: width, y: 3 * height / 4))  
25  
26         lines.setLineDash([10, 5], count: 2, phase: 0)  
27         lines.lineWidth = 5  
28         lines.stroke()  
29  
30  
31     }  
32 }
```

A red box highlights the last three lines of code: `lines.setLineDash([10, 5], count: 2, phase: 0)`, `lines.lineWidth = 5`, and `lines.stroke()`. A red arrow points from the text "Style and draw the line" to this highlighted area.

**Simulator:**

The iPhone 11 Pro Max simulator displays two horizontal dashed lines at approximately one-quarter and three-quarters of the screen height. The status bar shows the time as 3:17. The navigation bar indicates the app is "Running Session4Demos on iPhone 11 Pro Max". The bottom of the screen shows the Xcode interface with tabs for "Drawing", "Layers", "Animations 1", "Gestures", and "Animations 2".

The screenshot shows the Xcode interface with a Swift file named `DrawingView.swift` open. The code demonstrates the use of `UIBezierPath` to draw horizontal dashed lines and a large circular path.

```
16
17 // Draw dashed horizontal lines
18 let lines = UIBezierPath()
19
20 lines.move(to: CGPoint(x: 0, y: height / 4))
21 lines.addLine(to: CGPoint(x: width, y: height / 4))
22
23 lines.move(to: CGPoint(x: 0, y: 3 * height / 4))
24 lines.addLine(to: CGPoint(x: width, y: 3 * height / 4))
25
26 lines.setLineDash([10, 5], count: 2, phase: 0)
27 lines.lineWidth = 5
28 lines.stroke() Create a circular path
29
30 // Draw a circle
31 let circle = UIBezierPath(ovalIn: CGRect(x: width / 2 - 100,
32                                         y: height / 2 - 100,
33                                         width: 200,
34                                         height: 200))
35
36 UIColor.white.setFill()
37 circle.fill()
38
39 UIColor.purple.setStroke()
40 circle.lineWidth = 10
41 circle.stroke()
42
43
44
```

A red callout box with the text "Create a circular path" has an arrow pointing from line 28 to line 31. The iPhone 11 Pro Max simulator window shows a white screen with a large purple-outlined circle centered in the middle. At the bottom of the simulator window, there are several icons: Drawing, Layers, Animations 1, Gestures, and Animations 2.

# UIBezierPath

The screenshot shows the Xcode interface with a Swift file named DrawingView.swift open. The code demonstrates how to draw dashed horizontal lines and a filled circle using UIBezierPath.

```
16
17 // Draw dashed horizontal lines
18 let lines = UIBezierPath()
19
20 lines.move(to: CGPoint(x: 0, y: height / 4))
21 lines.addLine(to: CGPoint(x: width, y: height / 4))
22
23 lines.move(to: CGPoint(x: 0, y: 3 * height / 4))
24 lines.addLine(to: CGPoint(x: width, y: 3 * height / 4))
25
26 lines.setLineDash([10, 5], count: 2, phase: 0)
27 lines.lineWidth = 5
28 lines.stroke()
29
30 // Draw a circle
31 let circle = UIBezierPath(ovalIn: CGRect(x: width / 2 - 100,
32                                         y: height / 2 - 100,
33                                         width: 200,
34                                         height: 200))
35
36 UIColor.white.setFill()
37 circle.fill() ← Fill with white
38
39 UIColor.purple.setStroke()
40 circle.lineWidth = 10
41 circle.stroke()
42
43
44
```

The simulator window shows an iPhone 11 Pro Max displaying the results. It features two dashed horizontal lines at approximately one-quarter and three-quarters of the screen height, and a single large, solid purple circle centered at the bottom, representing the filled oval path.

The screenshot shows the Xcode interface with a Swift file named `DrawingView.swift` open. The code demonstrates how to draw dashed horizontal lines and a filled circle with a purple border.

```
16
17 // Draw dashed horizontal lines
18 let lines = UIBezierPath()
19
20 lines.move(to: CGPoint(x: 0, y: height / 4))
21 lines.addLine(to: CGPoint(x: width, y: height / 4))
22
23 lines.move(to: CGPoint(x: 0, y: 3 * height / 4))
24 lines.addLine(to: CGPoint(x: width, y: 3 * height / 4))
25
26 lines.setLineDash([10, 5], count: 2, phase: 0)
27 lines.lineWidth = 5
28 lines.stroke()
29
30 // Draw a circle
31 let circle = UIBezierPath(ovalIn: CGRect(x: width / 2 - 100,
32                                         y: height / 2 - 100,
33                                         width: 200,
34                                         height: 200))
35
36 UIColor.white.setFill()
37 circle.fill()
38
39 UIColor.purple.setStroke()
40 circle.lineWidth = 10
41 circle.stroke()
```

A red callout box highlights the last three lines of code (39-41) which set the stroke color to purple, set the line width to 10, and then stroke the path. A red arrow points from this callout to a red box containing the text "Add a purple border".

The iPhone 11 Pro Max simulator window shows the resulting drawing: two dashed horizontal lines at the top and bottom of the screen, and a large white circle with a thick purple border in the center.

The screenshot shows the Xcode interface with the Simulator window active. The top menu bar includes Simulator, File, Edit, Hardware, Debug, Window, Help, and a status bar showing 44% battery, Jan 25 3:38 PM, and a search icon. The bottom toolbar has icons for Stop, Run, Screenshot, and others.

The project navigation bar shows Session4Demos > iPhone 11 Pro Max. The code editor displays DrawingView.swift with the following content:

```
30 // Draw a circle
31 let circle = UIBezierPath(ovalIn: CGRect(x: width / 2 - 100,
32                                         y: height / 2 - 100,
33                                         width: 200,
34                                         height: 200))
35
36 UIColor.white.setFill()
37 circle.fill()
38
39 UIColor.purple.setStroke()
40 circle.lineWidth = 10
41 circle.stroke()
42
43 // Draw an arc
44 let arcCenter = CGPoint(x: width / 2, y: height / 2)
45 let arc = UIBezierPath(arcCenter: arcCenter,
46                       radius: 130,
47                       startAngle: 0,
48                       endAngle: CGFloat.pi * 3/2,
49                       clockwise: true)
50
51 UIColor.black.setStroke()
52 arc.lineWidth = 12
53 arc.lineCapStyle = .round
54 arc.stroke()
55 }
56 }
```

A red callout box with the text "Create a curved line" is positioned above the "arc.stroke()" line in the code, with a red arrow pointing downwards towards the simulator screen. The simulator shows an iPhone 11 Pro Max with a black dashed border. The screen displays a white background with a large, hollow purple circle in the center. A thick black curved line starts from the bottom right, goes up and around the left side of the purple circle, ending at the top right. The status bar on the simulator shows 3:38.

The screenshot shows the Xcode interface with a project named "Session4Demos" running on an iPhone 11 Pro Max simulator. The code in DrawingView.swift demonstrates how to draw a circle and an arc using UIBezierPath.

```
30 // Draw a circle
31 let circle = UIBezierPath(ovalIn: CGRect(x: width / 2 - 100,
32                                         y: height / 2 - 100,
33                                         width: 200,
34                                         height: 200))
35
36 UIColor.white.setFill()
37 circle.fill()
38
39 UIColor.purple.setStroke()
40 circle.lineWidth = 10
41 circle.stroke()
42
43 // Draw an arc
44 let arcCenter = CGPoint(x: width / 2, y: height / 2)
45 let arc = UIBezierPath(arcCenter: arcCenter,
46                       radius: 130,
47                       startAngle: 0,
48                       endAngle: CGFloat.pi * 3/2,
49                       clockwise: true)
50
51 UIColor.black.setStroke()
52 arc.lineWidth = 12
53 arc.lineCapStyle = .round
54 arc.stroke()
55 }
56 }
57 }
```

A red box highlights the code for styling and drawing the arc:

```
51 UIColor.black.setStroke()
52 arc.lineWidth = 12
53 arc.lineCapStyle = .round
54 arc.stroke()
```

A red callout arrow points from this highlighted code to a red box containing the text "Style and draw the arc".

The iPhone 11 Pro Max simulator shows the resulting drawing: a white circle with a thick purple stroke and a black-outlined arc starting from the bottom-left and ending at the top-right.

## REMINDER

- ▶ You must call `path.stroke()` or `path.fill()` for your bezier path to be rendered on screen

## HOW DOES IT WORK?

- ▶ **UIBezierPath** is a wrapper around **CGPath**, which comes from Core Graphics.
- ▶ Core Graphics is a lower-level framework. UIKit abstracts away some of the complexity.
  - ▶ Note: The “CG” prefix on **CGRect** and **CGPoint** stands for Core Graphics

# HOW DOES IT WORK?

- ▶ All drawing done by Core Graphics must take place inside a **CGContext**.

```
24 class CustomView: UIView {  
25  
26     func someMethod() {  
27         // this won't work!  
28         let path = UIBezierPath()  
29  
30         path.move(to: CGPoint(x: 100, y: 0))  
31         path.addLine(to: CGPoint(x: 100, y: 50))  
32  
33         UIColor.green.setStroke()  
34         path.stroke()  
35     }  
36 }
```



# HOW DOES IT WORK?

- ▶ The `draw` method implicitly provides a context for you.

```
override func draw(_ rect: CGRect) {  
    // create circle path  
    let circlePath = UIBezierPath(ovalIn: frame)  
  
    // set fill color  
    UIColor.blue.setFill()  
  
    // fill path  
    circlePath.fill()  
  
    let width = frame.size.width  
    let height = frame.size.height  
  
    // draw vertical line  
    let path = UIBezierPath()
```



## HOW DOES IT WORK?

- ▶ To trigger a re-draw of a view, do not call `draw` directly
  - ▶ Instead, call `myView.setNeedsDisplay()` if you need to tell your view to re-draw itself

BEAUTIFUL VIEWS WITH

---

LAYERS

# VISUAL PROPERTIES OF UIVIEW

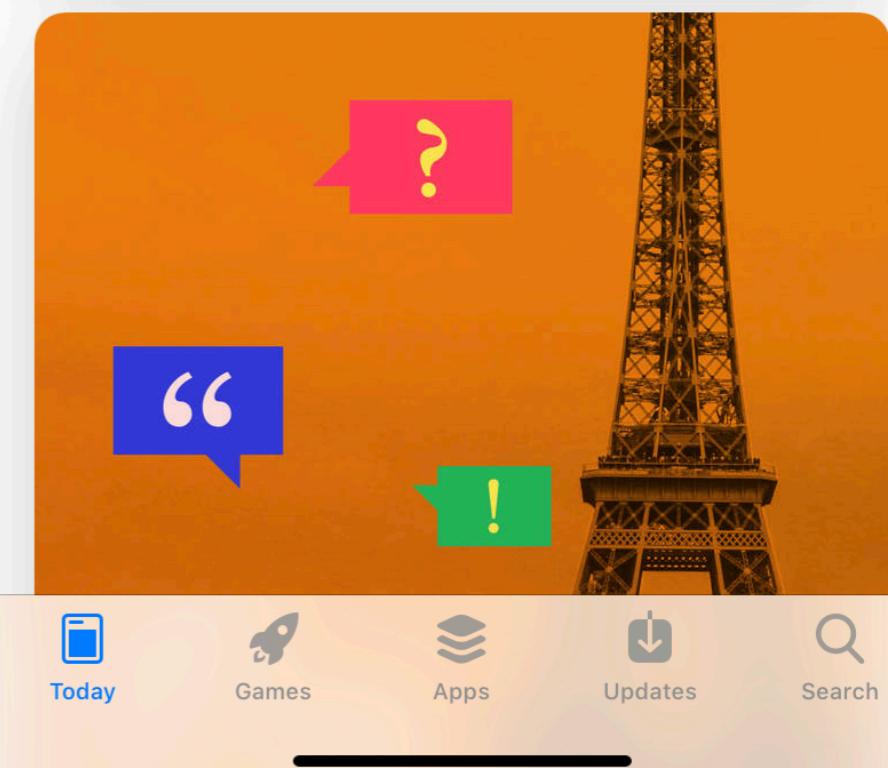
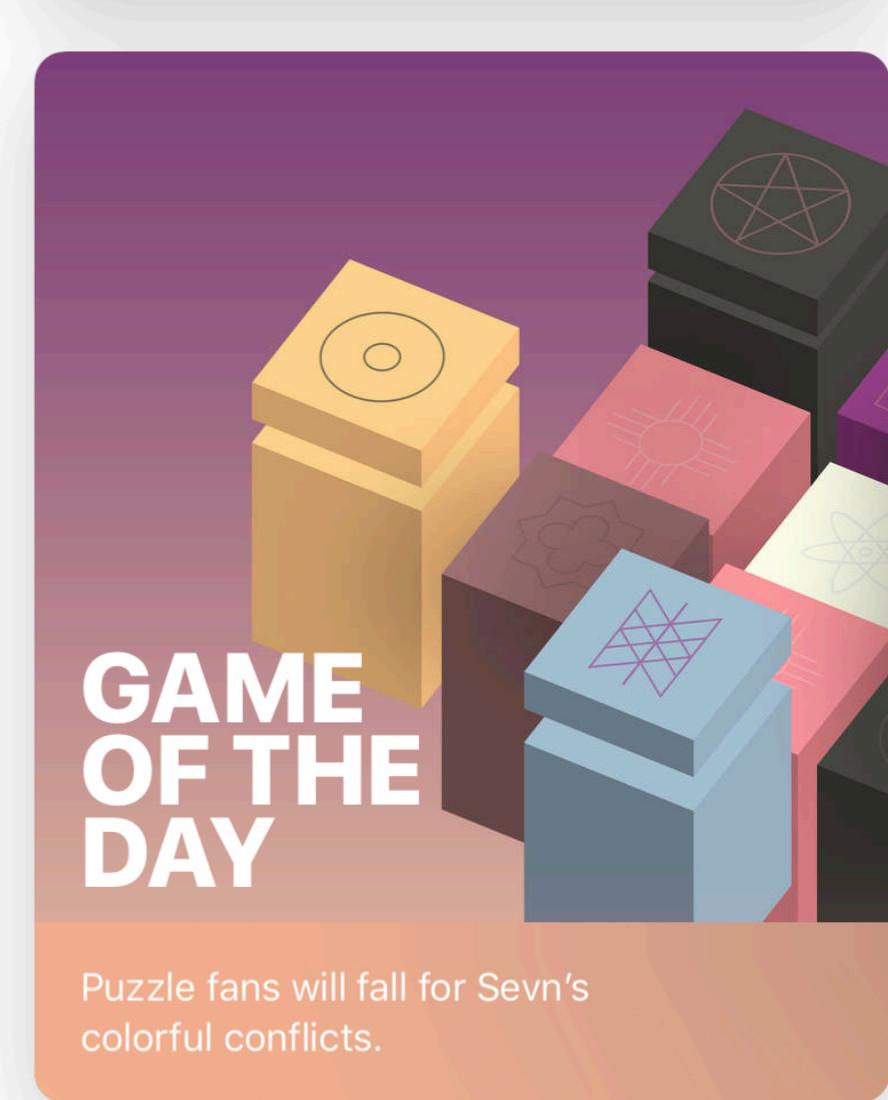
- ▶ So far, we've talked about:
  - ▶ Frame & bounds
  - ▶ Background color
  - ▶ Alpha

# CALayers

## WHAT ABOUT...

- ▶ Rounded corners?

App Store



## WHAT ABOUT...

► Shadows?

AirBnb

5:54



Try “Rome”

Dates

Guests

What can we help you  
find, Susan?



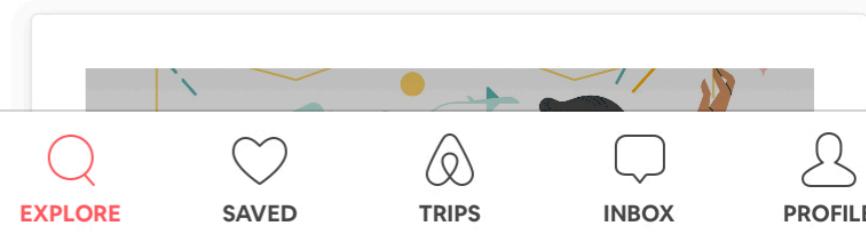
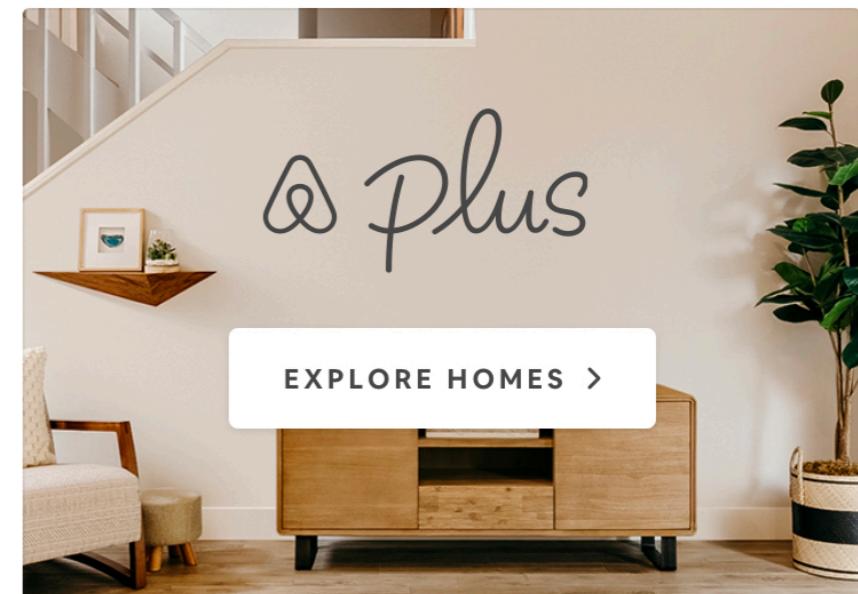
Homes



Experiences



Restaurants



# WHAT ABOUT...

- ▶ Gradients?

Calm

take a deep breath

## ... THESE ARE ALL PROPERTIES OF LAYERS

- ▶ Every UIView is backed by an instance of a class called CALayer
- ▶ Access a view's root layer with `view.layer`

# LAYERS

- ▶ Layers are similar to views:
  - ▶ Geometry of a layer is defined by **frame** & **bounds**
  - ▶ Appearance of a layer can be modified with properties like **backgroundColor**
  - ▶ A layer can have a **superlayer** and **sublayers**

# LAYERS

- ▶ Layers have a number of special properties that views do not, such as:
  - ▶ Corner radius
  - ▶ Border width and color
  - ▶ Shadow radius, offset and color

# LAYERS

- ▶ There are a number of subclasses of CALayer, such as:
  - ▶ **CATextLayer** for rendering strings
  - ▶ **CAShapeLayer** for drawing shapes
  - ▶ **CAGradientLayer** for creating gradients

# LAYERS

- ▶ Layers have many properties that can only be configured programmatically, not in storyboard
- ▶ A good place to do this is in `awakeFromNib`
  - ▶ This is a method on `UIView`
  - ▶ It can be overridden in a subclass
  - ▶ It's called after a view is instantiated from storyboard

The screenshot shows the Xcode interface with a code editor and a simulator window.

**Code Editor:**

```
8
9 import UIKit
10
11 class LayersView: UIView {
12
13     override func awakeFromNib() {
14         super.awakeFromNib()
15
16         layer.backgroundColor = UIColor.purple.cgColor
17         layer.cornerRadius = 20
18         layer.borderColor = UIColor.darkGray.cgColor
19         layer.borderWidth = 5
20     }
21 }
22
23
24
25
26
27
28
29
30
31
32
33
```

**Annotations:**

- A red callout box points from the text "Set cornerRadius to get rounded corners" to the line `layer.cornerRadius = 20`.
- A red callout box points from the text "Basic properties of CALayer" to the line `layer.backgroundColor = UIColor.purple.cgColor`.

**Simulator:**

The simulator displays an iPhone 11 Pro Max with a purple square view having rounded corners and a dark gray border. The status bar shows the time as 7:55. The bottom of the screen shows the Xcode toolbar with icons for Drawing, Layers, Animations 1, Gestures, and Animations 2.

The screenshot shows the Xcode interface with a Swift file open. The code defines a `LayersView` class that overrides `awakeFromNib()` to set the layer's background color to purple, border radius to 20, border color to dark gray, and border width to 5. A red box highlights the code that adds a shadow: `layer.shadowOpacity = 0.5`, `layer.shadowRadius = 3`, and `layer.shadowOffset = CGSize(width: 5, height: 5)`. An arrow points from the text "Add a shadow" at the bottom to this highlighted code. To the right, the iPhone 11 Pro Max simulator displays a purple rounded rectangle with a black border and a visible shadow.

```
8
9 import UIKit
10
11 class LayersView: UIView {
12
13     override func awakeFromNib() {
14         super.awakeFromNib()
15
16         layer.backgroundColor = UIColor.purple.cgColor
17         layer.cornerRadius = 20
18         layer.borderColor = UIColor.darkGray.cgColor
19         layer.borderWidth = 5
20
21         layer.shadowOpacity = 0.5
22         layer.shadowRadius = 3
23         layer.shadowOffset = CGSize(width: 5, height: 5)
24     }
25
26
27
28
29
30
31
32
33
```

Add a shadow

The screenshot shows the Xcode interface with a Swift file open. The code defines a `LayersView` class that overrides `awakeFromNib()`. Inside, it sets the view's layer properties like background color, corner radius, border color, and border width. It then adds a `CAGradientLayer` as a sublayer, setting its frame to the bounds of the view and defining a blue-to-white gradient with a 20px corner radius. A red callout box points from the text "Add a gradient sublayer" to the line where `gradientLayer.addSublayer(gradientLayer)` is called.

```
10
11 class LayersView: UIView {
12
13     override func awakeFromNib() {
14         super.awakeFromNib()
15
16         layer.backgroundColor = UIColor.purple.cgColor
17         layer.cornerRadius = 20
18         layer.borderColor = UIColor.darkGray.cgColor
19         layer.borderWidth = 5
20
21         layer.shadowOpacity = 0.5
22         layer.shadowRadius = 3
23         layer.shadowOffset = CGSize(width: 5, height: 5)
24
25         let gradientLayer = CAGradientLayer()
26         gradientLayer.frame = bounds
27         gradientLayer.colors = [UIColor.blue.cgColor,
28                               UIColor.white.cgColor]
29
30         gradientLayer.startPoint = CGPoint(x: 0, y: 0)
31         gradientLayer.endPoint = CGPoint(x: 1, y: 1)
32         gradientLayer.cornerRadius = 20
33         layer.addSublayer(gradientLayer)
34     }
35 }
```

## PART 1

---

# ANIMATIONS



## ANIMATIONS

## ANIMATION EXAMPLES

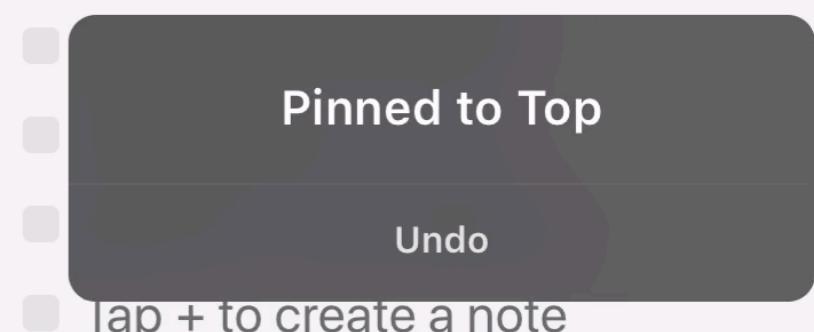
## ► Animate frame

Welcome to Jot, the place for all your quick notes, scraps, and ideas.

Try swiping this note to the left, to archive it.

View archived notes by searching.

Swipe any note to the right to copy or share it.



That's it — we hope you enjoy using Jot!



9:41



## ANIMATIONS

Search For A Book...



# ANIMATION EXAMPLES

## ► Animate alpha

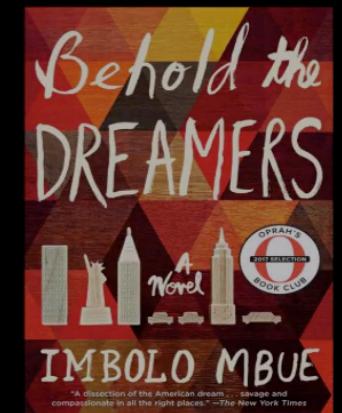
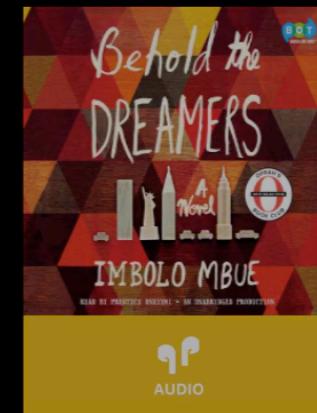
CHICAGO  
PUBLIC  
LIBRARY

eBooks, Audiobooks & More

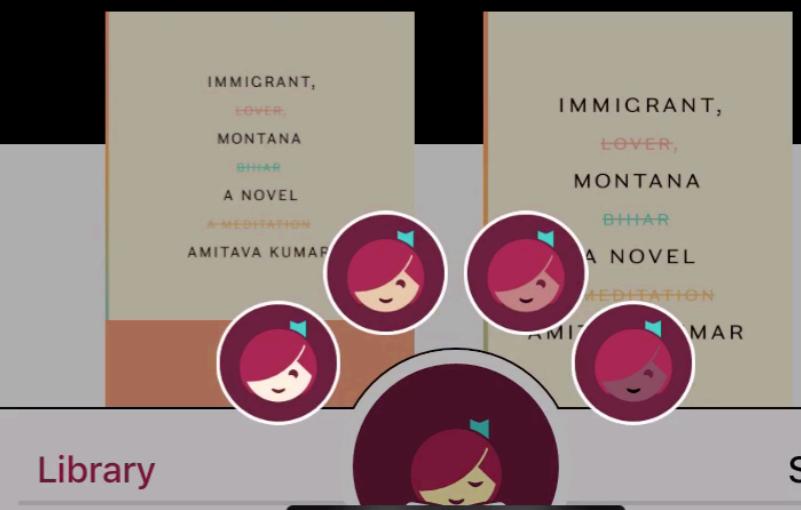
⊕ Preferences

Explore >

Immigrant Stories



Libby



Library

Shelf

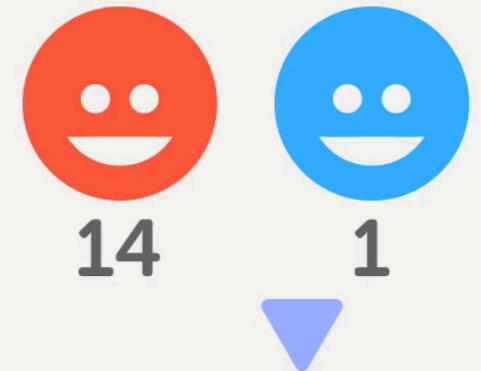
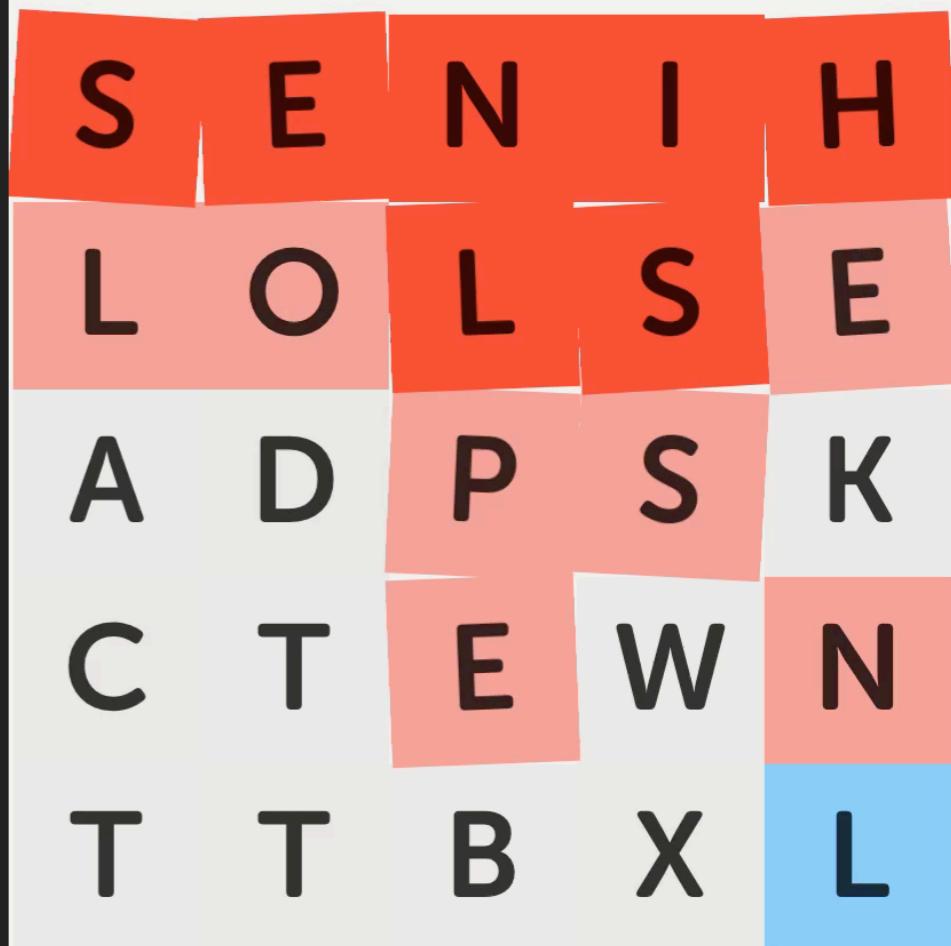


## ANIMATIONS

### ANIMATION EXAMPLES

- ▶ Animate transform (rotation)

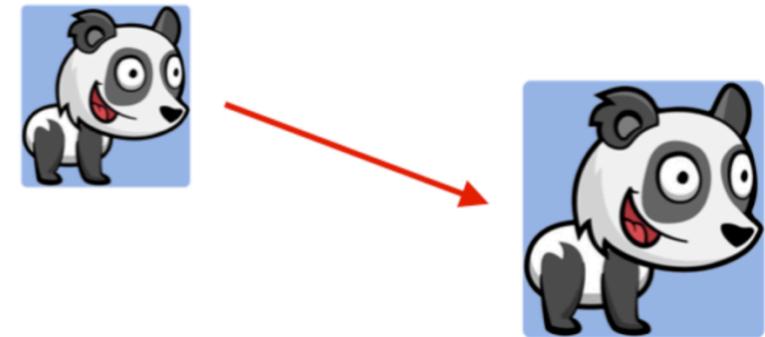
Letterpress



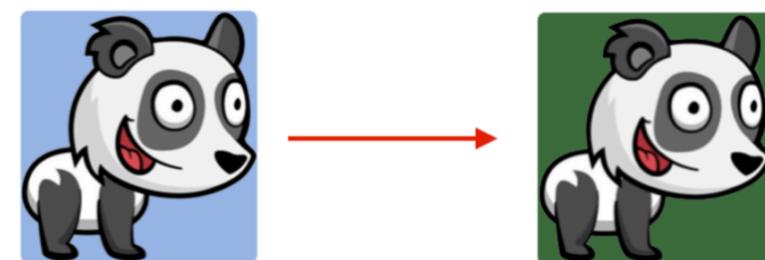
# ANIMATABLE PROPERTIES

- ▶ Only certain properties of UIView can be animated:
  - ▶ Bounds, frame, center
  - ▶ Background color
  - ▶ Alpha
  - ▶ Transform (e.g., scale and rotation)

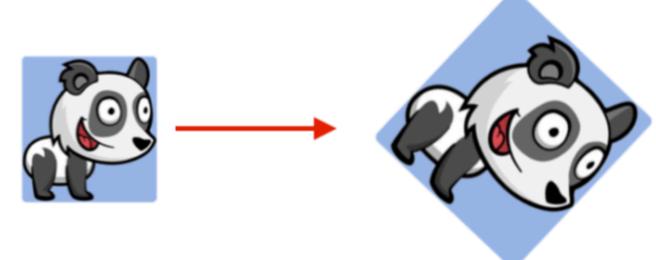
## Position and Size



## Appearance



## Transformation



# SIMPLE ANIMATION

The screenshot shows the Xcode interface with the following details:

- Top Bar:** Simulator, File, Edit, Hardware, Debug, Window, Help.
- Toolbar:** Standard Xcode icons.
- Project Navigator:** Session4Demos > Session4Demos > Animations 1 > Animations1ViewController.swift.
- Code Editor:** Shows the following Swift code:

```
7 //  
8  
9 import UIKit  
10  
11 class Animations1ViewController: UIViewController {  
12     @IBOutlet var pinkView: UIView!  
13  
14     @IBAction func tappedAnimate(_ sender: UIButton) {  
15         UIView.animate(withDuration: 1.0) {  
16             self.pinkView.alpha = 0.5  
17         }  
18     }  
19 }  
20 }  
21 }  
22 }  
23 }
```

A red callout arrow points from the text "The **animate** function takes a TimeInterval and a closure" to the line `UIView.animate(withDuration: 1.0)`.
- iPhone Simulator Preview:** An iPhone 11 Pro Max simulator window titled "Running Session4Demos on iPhone 11 Pro Max". It displays a red square with the text "Animate!" above it. The time is 8:25.
- Bottom Bar:** Drawing, Layers, Animations 1, Gestures, Animations 2.

# SIMPLE ANIMATION

The screenshot shows the Xcode interface with the following details:

- File menu:** Simulator, File, Edit, Hardware, Debug, Window, Help.
- Toolbar:** Standard Xcode toolbar icons.
- Project Navigator:** Session4Demos > Session4Demos > Animations 1 > Animations1ViewController.swift.
- Code Editor:** Displays the following Swift code:

```
8
9 import UIKit
10
11 class Animations1ViewController: UIViewController {
12
13     @IBOutlet var pinkView: UIView!
14
15     @IBAction func tappedAnimate(_ sender: UIButton) {
16
17         UIView.animate(withDuration: 1.0, delay: 0.25, options: [.curveEaseIn], animations: {
18             self.pinkView.alpha = 0.1
19         }, completion: { _ in
20             print("animation complete")
21         })
22     }
23 }
24
25 }
```

A red callout arrow points from the text "Pass in additional parameters to further customize the animation" to the line of code `options: [.curveEaseIn]`.
- Output Navigator:** Shows "Running Session4Demos on iPhone 11 Pro Max" with 5 warnings.
- iPhone 11 Pro Max Simulator:** Displays a pink square view and a blue "Animate!" button.

# SIMPLE ANIMATION

- ▶ Additional parameters you can pass in:
  - ▶ **Delay:** how long to wait before starting the animation
  - ▶ **Options:** special options like repeat, autoreverse, and animation curves
  - ▶ **Completion:** closure to execute when the animation finishes

## ANIMATION CURVES

- ▶ Specify whether the animation should slow down or speed up over the course of its run
- ▶ `.curveEaseInOut`: start slow, speed up, slow down
- ▶ `.curveEaseIn`: start slow, speed up, then stop
- ▶ `.curveEaseOut`: start fast, slow down, then stop
- ▶ `.curveLinear`: constant speed

## PROPERTY ANIMATORS

- ▶ iOS 10 introduced `UIViewControllerAnimated`
  - ▶ A new way of creating animations
  - ▶ The old way is “discouraged” but not deprecated... meaning, still widely in use

## PROPERTY ANIMATORS

- ▶ **UIViewControllerAnimated** allows fine-grained control over animations:
  - ▶ Pause and resume animations
  - ▶ Add additional animation blocks, even after an animation has begun
  - ▶ Modify the animation based on user interaction (swiping, dragging, etc.)

The screenshot shows the Xcode interface with a Swift file open. The code demonstrates two ways to perform animations: using `UIView.animate` and `UIViewControllerAnimated`.

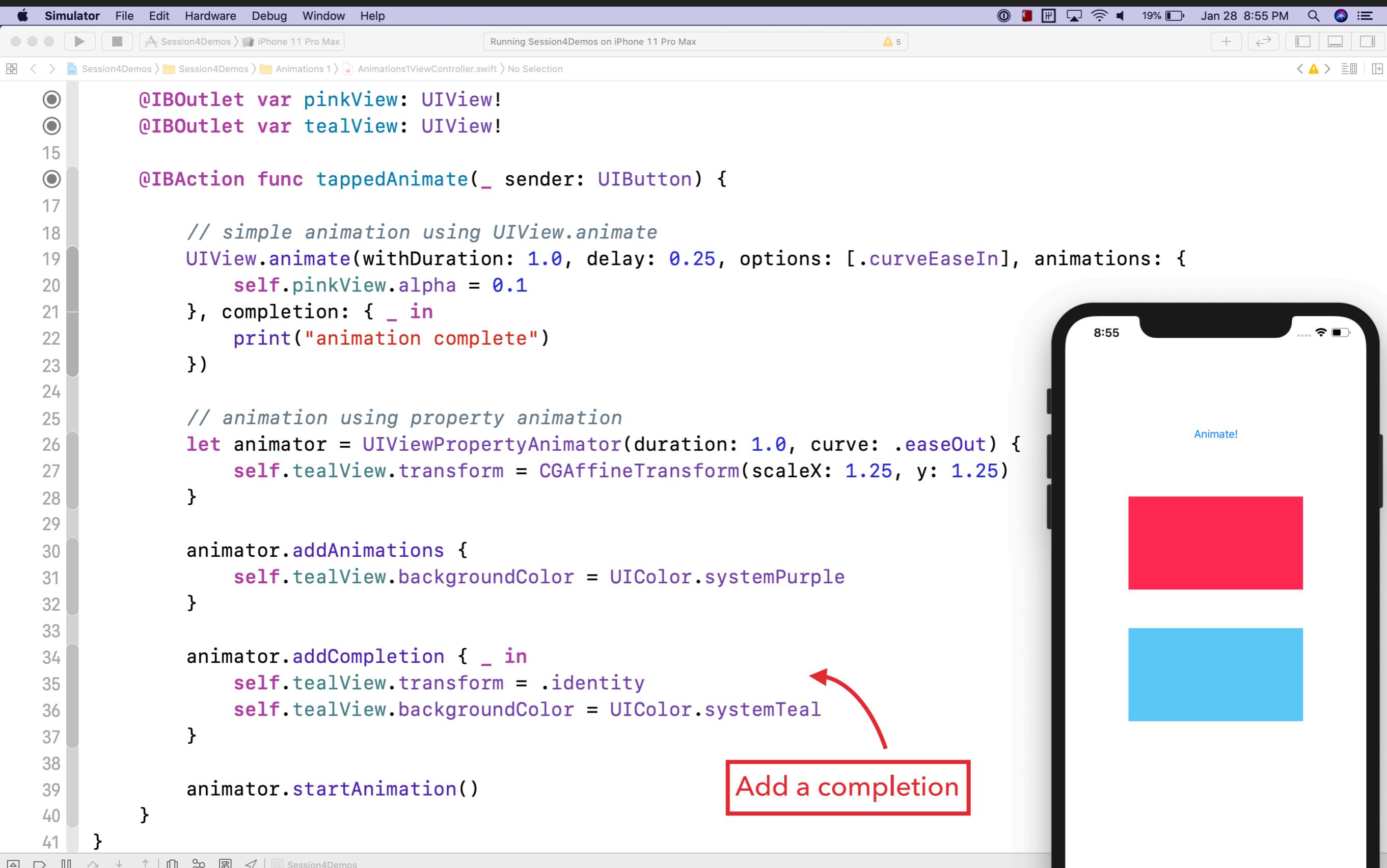
```
15 @IBOutlet var pinkView: UIView!
16 @IBOutlet var tealView: UIView!
17
18 // simple animation using UIView.animate
19 UIView.animate(withDuration: 1.0, delay: 0.25, options: [.curveEaseIn], animations: {
20     self.pinkView.alpha = 0.1
21 }, completion: { _ in
22     print("animation complete")
23 })
24
25 // animation using property animation
26 let animator = UIViewPropertyAnimator(duration: 1.0, curve: .easeOut) {
27     self.tealView.transform = CGAffineTransform(scaleX: 1.25, y: 1.25)
28 }
29
30 animator.addAnimations {
31     self.tealView.backgroundColor = UIColor.systemPurple
32 }
33
34 animator.addCompletion { _ in
35     self.tealView.transform = .identity
36     self.tealView.backgroundColor = UIColor.systemTeal
37 }
38
39 animator.startAnimation()
40 }
41 }
```

Annotations in red boxes point to specific parts of the code:

- A box labeled "Duration in seconds" points to the `duration: 1.0` parameter in the first `animate` call.
- A box labeled "Animation curve" points to the `options: [.curveEaseIn]` parameter in the first `animate` call.

To the right of the code, an iPhone 11 Pro Max simulator is shown running the application. It displays two views: a pink view at the top and a teal view below it. A blue button labeled "Animate!" is visible on the right side of the screen. The teal view is currently scaled up, demonstrating the effect of the property animation.





The screenshot shows the Xcode interface with a Swift file named `Animations1ViewController.swift` open. The code demonstrates two types of animations: a simple animation using `UIView.animate` and a property animation using `UIViewControllerAnimated`.

```
1 @IBOutlet var pinkView: UIView!
2 @IBOutlet var tealView: UIView!
3
4 @IBAction func tappedAnimate(_ sender: UIButton) {
5
6     // simple animation using UIView.animate
7     UIView.animate(withDuration: 1.0, delay: 0.25, options: [.curveEaseIn], animations: {
8         self.pinkView.alpha = 0.1
9     }, completion: { _ in
10        print("animation complete")
11    })
12
13
14     // animation using property animation
15     let animator = UIViewPropertyAnimator(duration: 1.0, curve: .easeOut) {
16         self.tealView.transform = CGAffineTransform(scaleX: 1.25, y: 1.25)
17     }
18
19     animator.addAnimations {
20         self.tealView.backgroundColor = UIColor.systemPurple
21     }
22
23     animator.addCompletion { _ in
24         self.tealView.transform = .identity
25         self.tealView.backgroundColor = UIColor.systemTeal
26     }
27
28     animator.startAnimation()
29 }
30
31 }
```

The iPhone simulator shows a screen with two views: a red view at the top and a blue view below it. A blue button labeled "Animate!" is on the right. A red arrow points from the text "Add a completion" to the completion block in the code.

The screenshot shows the Xcode interface with the following details:

- Top Bar:** Simulator, File, Edit, Hardware, Debug, Window, Help.
- Toolbar:** Standard Xcode icons for file operations.
- Project Navigator:** Session4Demos > iPhone 11 Pro Max.
- Assistant Editor:** Running Session4Demos on iPhone 11 Pro Max, showing 5 warnings.
- Code Editor:** Animations1ViewController.swift. The code demonstrates two types of animations: simple UIView.animate and property animation using UIViewPropertyAnimator. It includes outlets for pinkView and tealView, and actions for tapping a button to start an animation.
- Simulator Preview:** An iPhone 11 Pro Max simulator showing two views: a red view and a blue view. A button labeled "Animate!" is visible at the top right of the screen.

Annotations on the code:

- A red box highlights the line `animator.startAnimation()`.
- An arrow points from the text "Tell the animator to start" to this highlighted line.

```
1 @IBOutlet var pinkView: UIView!
2 @IBOutlet var tealView: UIView!
3
4
5 @IBAction func tappedAnimate(_ sender: UIButton) {
6
7     // simple animation using UIView.animate
8     UIView.animate(withDuration: 1.0, delay: 0.25, options: [.curveEaseIn], animations: {
9         self.pinkView.alpha = 0.1
10    }, completion: { _ in
11        print("animation complete")
12    })
13
14
15     // animation using property animation
16     let animator = UIViewPropertyAnimator(duration: 1.0, curve: .easeOut) {
17         self.tealView.transform = CGAffineTransform(scaleX: 1.25, y: 1.25)
18     }
19
20     animator.addAnimations {
21         self.tealView.backgroundColor = UIColor.systemPurple
22     }
23
24     animator.addCompletion { _ in
25         self.tealView.transform = .identity
26         self.tealView.backgroundColor = UIColor.systemTeal
27     }
28
29
30     animator.startAnimation() ← Tell the animator to start
31
32 }
33
34 }
```

ADDING USER INTERACTION

---

GESTURE RECOGNIZERS

# GESTURE RECOGNIZERS

- ▶ Gesture recognizers are objects that allow your code to detect and respond to common gestures:
  - ▶ Tap
  - ▶ Pinch
  - ▶ Rotation
  - ▶ Swipe
  - ▶ Pan
  - ▶ Screen edge pan
  - ▶ Long press

# GESTURE RECOGNIZERS

- ▶ **UIGestureRecognizer** is a base class that has concrete subclasses for each possible gesture

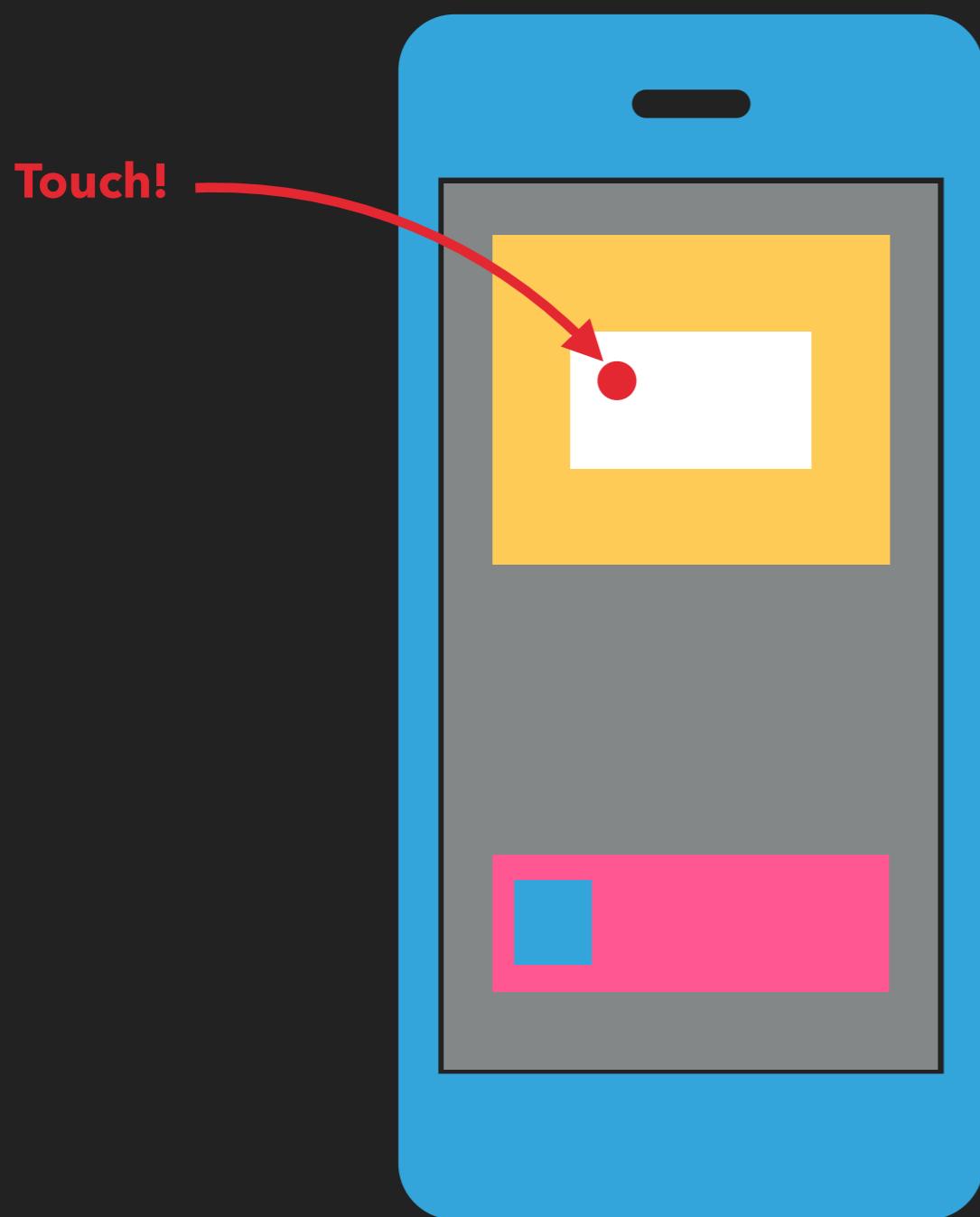
The concrete subclasses of **UIGestureRecognizer** are the following:

- [UITapGestureRecognizer](#)
- [UIPinchGestureRecognizer](#)
- [UIRotationGestureRecognizer](#)
- [UISwipeGestureRecognizer](#)
- [UIPanGestureRecognizer](#)
- [UIScreenEdgePanGestureRecognizer](#)
- [UILongPressGestureRecognizer](#)

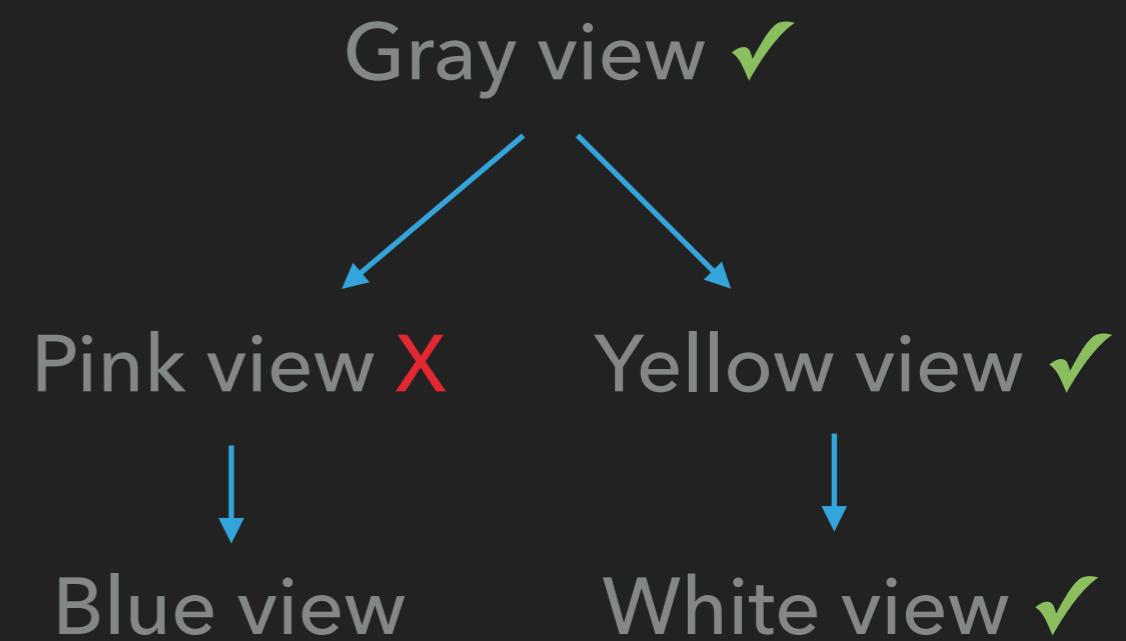
# GESTURE RECOGNIZERS

- ▶ **Hit-testing** is used to determine which view should receive a touch event from the user
  - ▶ The user touches the screen at a particular point
  - ▶ The system traverses the view hierarchy, starting with the root view, looking for the front-most view containing the point

# GESTURE RECOGNIZERS



## EXAMPLE: HIT-TESTING VIEWS



# GESTURE RECOGNIZERS

- ▶ Each gesture recognizer you create must be **associated with a specific view**
  - ▶ The gesture recognizer responds to touch events for that view and its subviews
  - ▶ If a view receives a touch event that it can't respond to, the system will traverse down the view hierarchy looking for a superview to handle the event

# GESTURE RECOGNIZERS

- ▶ Gesture recognizers are initialized with a target and an action (selector)

```
10  
11 class GesturesViewController: UIViewController {  
12  
13     override func viewDidLoad() {  
14         super.viewDidLoad()  
15  
16         let tapGestureRecognizer = UITapGestureRecognizer(target: self,  
17                                                 action: #selector(tap))  
18         tapGestureRecognizer.numberOfTapsRequired = 2  
19         view.addGestureRecognizer(tapGestureRecognizer)  
20     }  
21 }
```

# GESTURE RECOGNIZERS

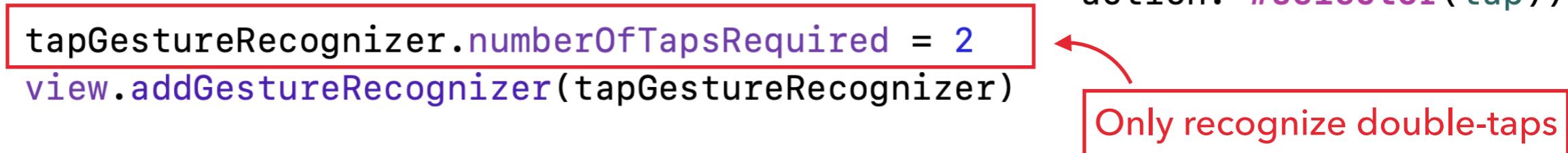
- ▶ Call `addGestureRecognizer` to associate the gesture recognizer with a specific view

```
10  
11 class GesturesViewController: UIViewController {  
12  
13     override func viewDidLoad() {  
14         super.viewDidLoad()  
15  
16         let tapGestureRecognizer = UITapGestureRecognizer(target: self,  
17                                               action: #selector(tap))  
18         tapGestureRecognizer.numberOfTapsRequired = 2  
19         view.addGestureRecognizer(tapGestureRecognizer)  
20     }  
21 }
```

# GESTURE RECOGNIZERS

- ▶ Gesture recognizers also have configurable properties, depending on the type of gesture

```
10  
11 class GesturesViewController: UIViewController {  
12  
13     override func viewDidLoad() {  
14         super.viewDidLoad()  
15  
16         let tapGestureRecognizer = UITapGestureRecognizer(target: self,  
17                                                 action: #selector(tap))  
18         tapGestureRecognizer.numberOfTapsRequired = 2  
19         view.addGestureRecognizer(tapGestureRecognizer)  
20     }  
21 }
```



Only recognize double-taps

# TAP GESTURE RECOGNIZER

## Configuring the Gesture

`var numberOfTapsRequired: Int`

The number of taps for the gesture to be recognized.

`var numberOfTouchesRequired: Int`

The number of fingers required to tap for the gesture to be recognized.

# SWIPE GESTURE RECOGNIZER

## Configuring the Gesture

`var direction: UISwipeGestureRecognizer.Direction`  
The permitted direction of the swipe for this gesture recognizer.

`var numberOfTouchesRequired: Int`  
The number of touches that must be present for the swipe gesture to be recognized.

# PAN GESTURE RECOGNIZER

## Configuring the Gesture Recognizer

`var maximumNumberOfTouches: Int`

The maximum number of fingers that can be touching the view for this gesture to be recognized.

`var minimumNumberOfTouches: Int`

The minimum number of fingers that can be touching the view for this gesture to be recognized.

# SCREEN EDGE PAN GESTURE RECOGNIZER

## Specifying the Starting Edges

```
var edges: UIRectEdge
```

The acceptable starting edges for the gesture.

# LONG PRESS GESTURE RECOGNIZER

## Configuring the Gesture Recognizer

`var minimumPressDuration: TimeInterval`

The minimum period fingers must press on the view for the gesture to be recognized.

`var numberOfTouchesRequired: Int`

The number of fingers that must be pressed on the view for the gesture to be recognized.

`var numberOfTapsRequired: Int`

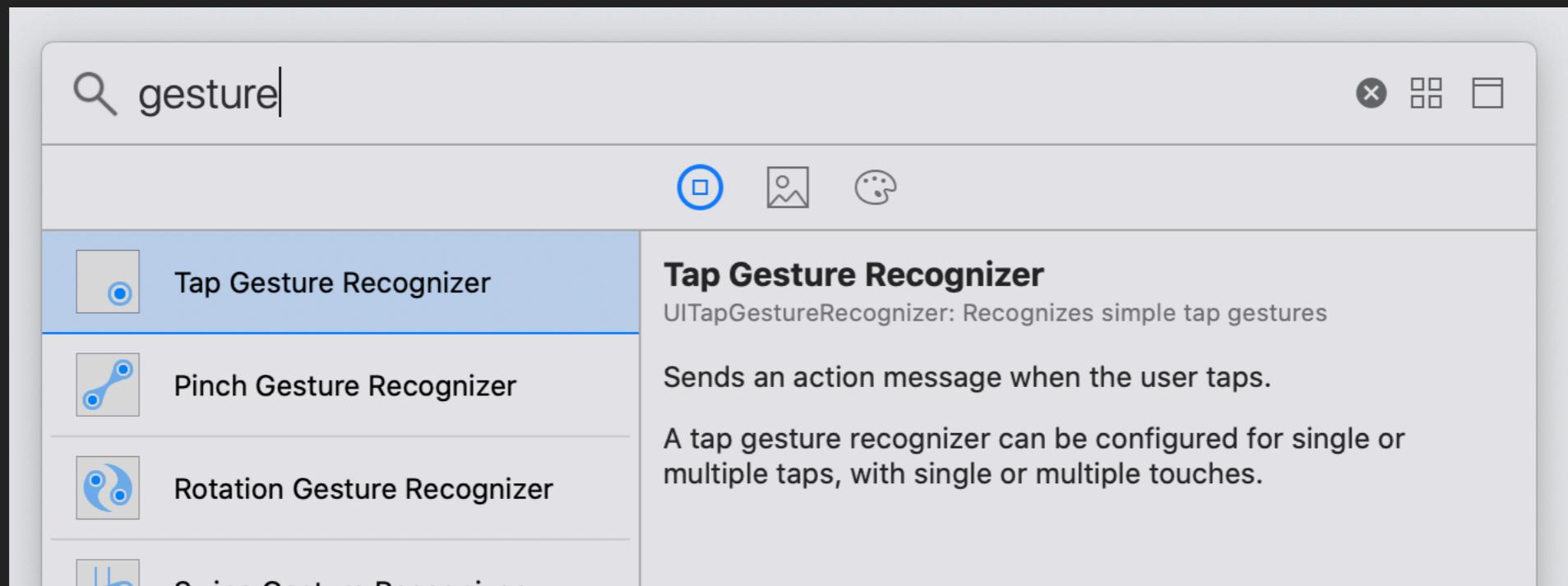
The number of taps on the view required for the gesture to be recognized.

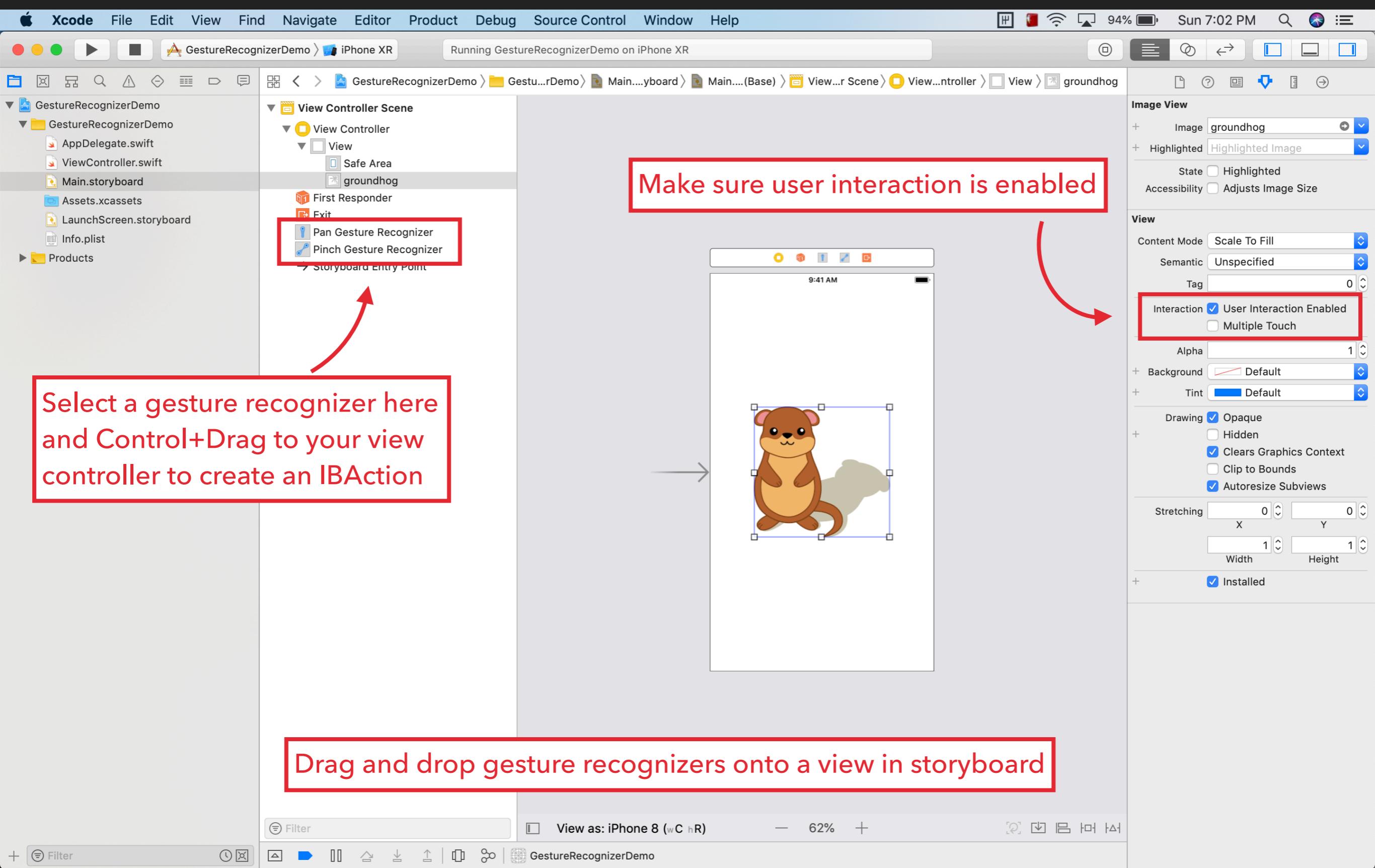
`var allowableMovement: CGFloat`

The maximum movement of the fingers on the view before the gesture fails.

# GESTURE RECOGNIZERS

- ▶ Gesture recognizers can also be added and configured in storyboard
  - ▶ Drag and drop from Object Library
  - ▶ Connect to your Swift code using IBAction





# GESTURE RECOGNIZERS

- ▶ Hook up gesture recognizers to code with IBActions

```
9 import UIKit
10
11 class ViewController: UIViewController {
12
13     @IBAction func pan(_ sender: UIPanGestureRecognizer) {
14         print("Pan!")
15     }
16
17     @IBAction func pinch(_ sender: UIPinchGestureRecognizer) {
18         print("Pinch!")
19     }
20 }
21
```

## HANDLING GESTURE RECOGNIZERS

- ▶ Many gestures are **continuous**, meaning they take place over a period of time
- ▶ The **action** method will be called multiple times
- ▶ You can inspect properties of the gesture recognizer to determine how to respond each time the method is called

# GESTURE RECOGNIZERS

- ▶ You may want to update your view based on the location of the touch or the number of touches

## Getting the Touches and Location of a Gesture

`func location(in: UIView?) -> CGPoint`

Returns the point computed as the location in a given view of the gesture represented by the receiver.

`func location(ofTouch: Int, in: UIView?) -> CGPoint`

Returns the location of one of the gesture's touches in the local coordinate system of a given view.

`var numberOfTouches: Int`

Returns the number of touches involved in the gesture represented by the receiver.

## GESTURE RECOGNIZERS

- ▶ You may also want to know the **state** of the gesture recognizer, using `gestureRecognizer.state`
- ▶ Possible values include:
  - ▶ Began
  - ▶ Changed
  - ▶ Ended
  - ▶ Possible

## GESTURE RECOGNIZERS

- ▶ Specific subclasses of `UIGestureRecognizer` will also have their own properties
- ▶ For example:
  - ▶ Pan – `translation` and `velocity`
  - ▶ Pinch – `scale` and `velocity`

## SIMULATOR TESTING

- ▶ How do you simulate gestures that require two fingers?
  - ▶ Hold down **Option** to get two “fingers” on simulator
  - ▶ Hold down **Option+Shift** to move both “fingers” together

## PART 2

---

# ANIMATIONS

Three yellow starburst icons arranged in a triangular pattern, pointing upwards and to the right.

## BUILDING MORE COMPLEX ANIMATIONS

- ▶ We have seen how to animate properties of `UIView`
- ▶ You can also animate properties of `CALayer` to create more complex animations
  - ▶ `CALayer` is part of the `Core Animation` framework

## BUILDING MORE COMPLEX ANIMATIONS

- ▶ See CoreAnimationDemo project in class repo for code samples

# BUILDING MORE COMPLEX ANIMATIONS

- ▶ Subclasses of CALayer
  - ▶ CAGradientLayer
  - ▶ CAShapeLayer
  - ▶ CATextLayer

# BUILDING MORE COMPLEX ANIMATIONS

## ▶ CABasicAnimation

- ▶ Animate a property from one value to another
- ▶ E.g., use **strokeEnd** to animate drawing a path

CABasicAnimation



CAKeyframeAnimation

CASpringAnimation

CATransition

# BUILDING MORE COMPLEX ANIMATIONS

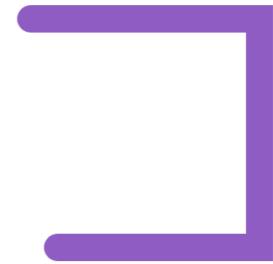
## ▶ CAKeyframeAnimation

- ▶ Animate a property between multiple values
- ▶ E.g., animate **strokeColor** from pink to purple to blue

CABasicAnimation



CAKeyframeAnimation



CASpringAnimation

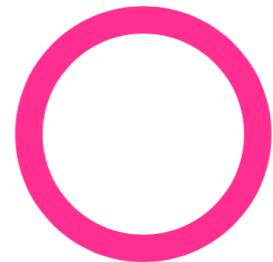
CATransition

# BUILDING MORE COMPLEX ANIMATIONS

## ► CASpringAnimation

- ▶ Animate a property with a spring (bounce) effect
- ▶ E.g., use spring animation on `transform.scale.y`

CABasicAnimation



CAKeyframeAnimation



CASpringAnimation

Hello

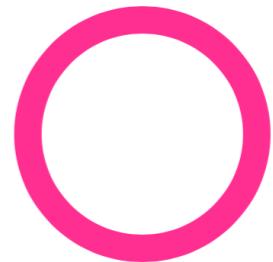
CATransition

# BUILDING MORE COMPLEX ANIMATIONS

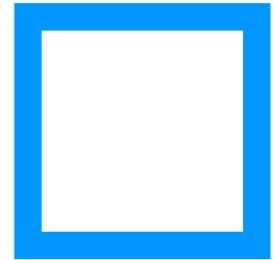
## ► CATransition

- ▶ Animate a layer with a given transition style
- ▶ E.g., initial state floats to the top as a new state appears

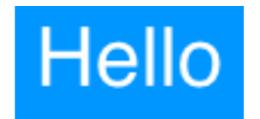
CABasicAnimation



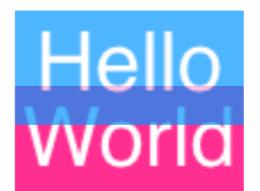
CAKeyframeAnimation



CASpringAnimation



CATransition



ASSIGNMENT #4

---

TIC TAC TOE

# TIC TAC TOE

- ▶ Build a Tic Tac Toe game for two players to play against each other on a single device
- ▶ You will use:
  - ▶ `UIBezierPath`
  - ▶ `CALayers`
  - ▶ `UIView` animation
  - ▶ `UIGestureRecognizers`

## TIC TAC TOE: TIP #1

- ▶ You will need to determine when the user releases a piece over a particular square on the grid
- ▶ If you place nine transparent views on the grid, you can check if the piece overlaps with a square by calling:

`piece.frame.intersects( square.frame )`

The screenshot shows the Xcode interface with a project named "TicTacToeStarter". The storyboard displays a 3x3 grid of squares, with the top-left square containing an "X" and the bottom-right square containing an "O". The code editor shows the `ViewController.swift` file:

```
// TicTacToeStarter
//
// Created by Susan Stevens on 1/30/19.

import UIKit

class ViewController: UIViewController {

    @IBAction func movedPiece(_ sender: UIPanGestureRecognizer) {
        guard let piece = sender.view else { return }

        let translation = sender.translation(in: view)
        piece.frame.origin.x += translation.x
        piece.frame.origin.y += translation.y

        sender.setTranslation(CGPoint(x: 0, y: 0), in: view)
    }
}
```

A red callout box highlights the line `squares` in the storyboard connection inspector, with the text: "Select 'Outlet Collection' to create one outlet that will hold all nine squares". A red arrow points from this text to the `squares` field in the inspector.

The screenshot shows the Xcode interface with a storyboard and associated Swift code.

**Storyboard View:** The storyboard displays a "Tic Tac Toe" game interface. At the top, it says "9:41 AM". Below is a title "Tic Tac Toe". A 3x3 grid of squares is shown. The square at index 1,1 (top-middle) is highlighted in blue and labeled "Square1". A blue dot on the storyboard indicates a connection to the "squares" outlet. The bottom row contains a blue "X" and a pink "O".

**File Navigator:** Shows the project structure: TicTacToeStarter > TicT...arter > Main.storyboard > MainScene > View Controller > Square0.

**Code Editor:** The code is in ViewController.swift:

```
3 // TicTacToeStarter
4 //
5 // Created by Susan Stevens on 1/30/19.
6 // Copyright © 2019 Susan Stevens. All rights reserved.
11 class ViewController: UIViewController {
12
13 @IBOutlet var squares: [UIView]!
14
15 @IBAction func movedPiece(_ sender: UIPanGestureRecognizer) {
16     guard let piece = sender.view else { return }
17
18     let translation = sender.translation(in: view)
19     piece.frame.origin.x += translation.x
20     piece.frame.origin.y += translation.y
21
22     sender.setTranslation(CGPoint(x: 0, y: 0), in: view)
23 }
24
25 }
```

A red callout box highlights the line "@IBOutlet var squares: [UIView]!" with the text: "Outlet Collection" gives you an array to hold multiple views. A red arrow points from this callout to the line of code.

## TIC TAC TOE: TIP #2

- ▶ There will be a lot going on in your view controller (handling gestures and animations)
- ▶ It's a good idea to isolate logic for determining the current state of the grid into its own class

```
class ViewController: UIViewController {
```

```
    let grid = Grid()
```

```
    . . .
```

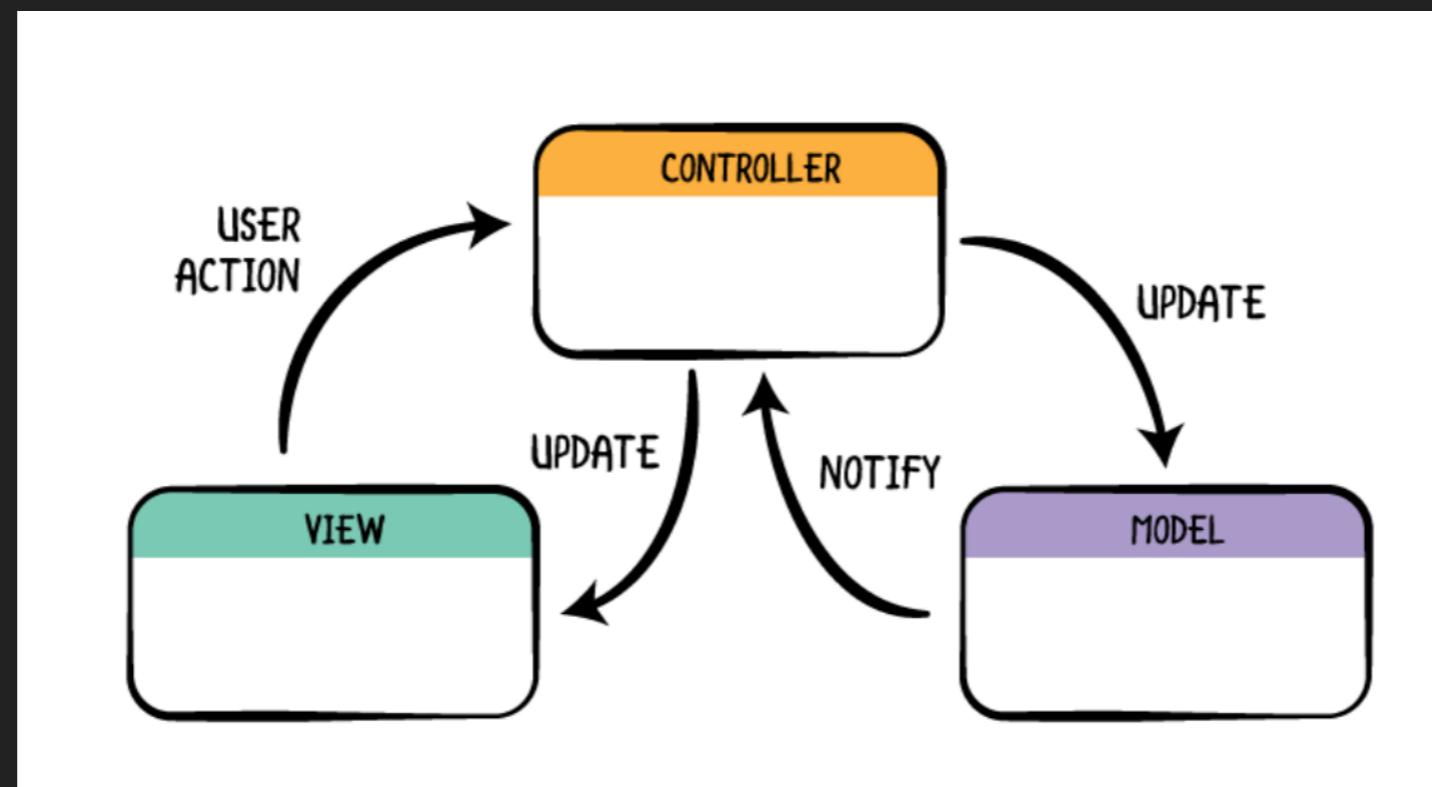
## TIC TAC TOE: TIP #2

- ▶ What information about the **state of the game** will the view controller need from **grid**?
- ▶ What information about the **view** will the view controller need to provide to **grid**?

# MVC

- ▶ MVC = Model-View-Controller
- ▶ The #1 design pattern in iOS

REMEMBER THIS?



# MVC

- ▶ Model classes we have seen so far:
  - ▶ HW 2: **Animal**
  - ▶ HW 3: **GitHubIssue**
  - ▶ HW 4: **Grid**

# TIC TAC TOE

- ▶ Due Wednesday, February 5 at 5:29pm
- ▶ Post any questions to Slack